A Framework for the Design of Parallel Adaptive Libraries on Hard Computational Problems

Alfredo Goldman, Yanik Ngoko, Denis Trystram

A Framework for the Design of Parallel Adaptive Libraries on Hard Computational Problems

Alfredo Goldman[1] and Yanik Ngoko[1] ⋆ and Denis Trystram[2]

[1] DCC - IME - USP Rua do Matão, 1010
CEP 05508-090 São Paulo - Brazil
yanik@ime.usp.br, gold@ime.usp.br
[2] Grenoble University, ENSIMAG
681, rue de la passerelle
Domaine universitaire - BP 72 - 38402 SAINT MARTIN D'HERES
denis.trystram@imag.fr

Abstract. In this work, we present the Adaptive Multi-Selection Framework (called AMF). AMF is an API built for helping designers to develop optimized combinations of multiple algorithms solving the same problem in function of the physical architecture and algorithm behavior. AMF offers a simple and generic model for developing automatic combination of algorithms. In this model, the user needs to specify the set of algorithms to be combined and a representative benchmark of instances of the problem solved by the algorithms. This generic solution has advantages over many existing solutions for making automatic combination that are specific to a fixed set of algorithms or computational problems. Automatic combinations of algorithms are made in AMF with the multi-selection technique. For each instance of a computational problem, its resolution under multi-selection includes a selection of a subset of candidate algorithms followed by a concurrent run of the selected algorithms with a smart resource sharing. The resource sharing is decided according to the physical architecture, the problem instance and the time allowed to compute it. The multi-selection strategy provides excellent results when there is a large variance of execution time per instance. The actual implementation of AMF is built for shared memory architectures. However, it can be extended to distributed ones. The AMF principles have been validated in particular on the well-known Constraint Satisfaction Problem.

Keywords: adaptive algorithms, adaptive systems, automatic algorithm combination, resource sharing

1 Introduction

The continuous evolution of algorithmics is leading to a huge amount of algorithms available for each computational problem. For the same problem, the performance of these algorithms might vary depending on many aspects like the considered problem instance or the machine architecture [1, 19]. In order

⋆ Corresponding author

to obtain good performances, there is the need for solutions that can combine efficiently various algorithms designed for solving the same problem. The huge variety of computational problems for which such solutions are required, the large amount of algorithms and machine architectures suggest that automatic generic combinations should be prioritized. In this paper, we focus on the automatic combination of multiple algorithms solving the same problem, specially those related to hard computational problems. Our main objective is to provide a framework that eases the implementation of automatic combination of algorithms.

1.1 Contributions

We propose the Adaptive Multi-Selection Framework designed to ease the task of combining multiple algorithms solving the same problem on parallel architectures. For this end, AMF typically requires for each computational problem a set of candidate algorithms (sequential or parallel) and a benchmark for tuning algorithm performances. These information determine AMF knowledge on building automatic combination of algorithms for the targeted computational problem.

AMF works like a problem solver and provides an interface where any instance of a computational problem can be solved under the multi-selection technique. The resolution of a problem instance under the multi-selection contains three phases: a selection of a set of candidate algorithms for the instance, a computation of an optimal resource sharing for the instance and the execution of the algorithm combination related to this selection. These combinations are defined in AMF as algorithms portfolio [12]. An algorithm portfolio execution is a set of algorithms run each with a predefined number of resources an stopped as soon as one algorithm completes its execution.

The actual version of AMF is designed for shared memory architectures and support parallelism based on threads. Moreover, we provide some validations of its utilization for defining automatic combination of algorithms for the Constraint Satisfaction Problem, either sequential or parallel.

1.2 Text organization

The rest of the paper is organized as follows: Section 2 presents the multi-selection technique as designed in AMF. The architecture of AMF in a component point of view is presented in Section 3. Section 4 gives an example of utilization of AMF on the Constraint Satisfaction Problem. In Section 5, we discuss about advantages of the multi-selection. Related works are presented in Section 6 and we conclude in Section 7.

2 The Multi-selection technique

We suppose that we have a parallel machine architecture and a finite set of algorithms(parallel or sequential) solving a same problem P. Within multi-selection,

each instance of P is solved in three phases. The first phase consists in a selection of candidate algorithms for the instance. The second phase consists in sharing resources between the selected algorithms. The third phase is a concurrent execution of the selected algorithms under the adopted resource sharing. Each phase is described in the following.

2.1 Selection of the candidate algorithms

At this stage, we have a finite set of algorithms (parallel or sequential) solving the computational problem $\mathcal{P}$. Given an instance of $\mathcal{P}$, we have to decide which algorithm to use. The multi-selection considers two modes for instance resolution: the *online* and *offline* mode. These two modes affect differently the selection phase.

Let suppose that we have a base $\mathcal{A}$ of candidate algorithms for solving an instance I of $\mathcal{P}$. In the offline mode, all candidate algorithms known for $\mathcal{P}$ will be selected at this stage of the multi-selection. Thus the selection in the offline mode will output $\mathcal{A}(I) = \mathcal{A}$. In the online mode, just a subset of candidate algorithms is retained. This means that the selection in the online mode will output a set $\mathcal{A}(I)$ where $\mathcal{A}(I) \subseteq \mathcal{A}$.

Details about the selection phase in AMF will be given in Section 3.1. For now, we can retain that the two possible modes (online and offline) lead to different types of overhead on instance resolution. We illustrate this as follows: let us denote the total resolution time for solving I as $t(I)$. In multi-selection, we have $t(I) = t_s(I)+t_{rs}(I)+t_{ep}(I)$ where, t_s, t_{rs} and t_{ep} are, respectively, the time for selecting a subset of algorithms, the time for computing a resource sharing, and the time for executing the chosen algorithms with the computed resource sharing. Since in the offline mode the selection of algorithms is the same for each problem instance, one can pre-compute the optimal resource sharing that will be re-used for all instances. Thus, the cost $t_s(I) + t_{rs}(I)$ will typically be negligible in the offline mode.

Algorithms selected at this phase of the multi-selection will then be executed concurrently. However, it is only the result of one execution that will be exploited at the end. This means that the more selected algorithms, the more the overhead in the resolution. The philosophy of the online mode is to try to reduce $t_{ep}(I)$ by efficient selection of algorithms even if it will lead to more significant values of $t_s(I) + t_{rs}(I)$. Such a selection can be guide by a comparative analysis of instance feature and algorithm behavior [9].

2.2 Computation of the optimal resource sharing

For the determination of resource sharing (second phase in multi-selection), the multi-selection uses the *dRSSP* model [5]. In *dRSSP*, we assume that we have a finite set of homogeneous computation units or resources $P = \{0, \ldots, m\}$. On these units, parallel algorithms can be run.

Given a computational problem $\mathcal{P}$, the inputs for *dRSSP* model are : a finite set of algorithms $\mathcal{A} = \{A_1, \ldots A_k\}$, a finite set $\mathcal{I} = \{I_1, \ldots, I_n\}$ of representative

instances of $\mathcal{P}$, and cost values $C(A_i, I_j, p)$ giving the execution time of each algorithm $A_i \in \mathcal{A}$ on the instance $I_j \in \mathcal{I}$ when executed on $p \in P$ resources.

The resolution of instances in the *dRSSP* model is based on Algorithm portfolio [12]. Let us define a resource sharing as a vector $S = (S_1, \ldots, S_k)$ such that $S_i \in P$ and $\sum S_i \leq m$. Here, S_i is the number of resources used by A_i. With Algorithm portfolio, we can define the resolution time of any instance under this resource sharing as $C(S, I_j) = \min_{A_i \in \mathcal{A}} \{C(A_i, I_j, S_i) | S_i > 0)\}$ [3].

Given these inputs, we suppose in *dRSSP* that to solve the problem $\mathcal{P}$, any of its instance will behave like one instance in $\mathcal{I}$. Therefore, a global approach to minimize the resolution time of the problem instance can consist in finding the resource sharing S minimizing $\sum_{I_j \in \mathcal{I}} C(S, I_j)$. We will denote this as the **MinSum** optimization function.

In the optimal solution under the MinSum objective, significant variations can be observed between instances resolution time. In a competition setting where we have a finite set of instances to solve in a maximal amount of time, this might not be a problem. However, in a context where instances are not solved in block, variations between execution times of instances are sensitive. In this case for example, a good optimization goal is to minimize the maximal time we can wait for having the solution of an instance. This will be taken as the **MinMax** objective given by the function: minimize $\max_{I_j \in \mathcal{I}} C(S, I_j)$.

Under the MinMax or MinSum objectives, one can easily show that the problem of computing the optimal resource sharing is NP complete [5]. Thus, heuristics must be used in the online mode in order to have an acceptable overhead on instance resolution (in the offline mode the resource sharing is pre-computed).

Minimize $\sum_{I_j \in \mathcal{I}} C(S, I_j)$
1. $S_i \in \{0, \ldots, m\}$
2. $\sum S_i \leq m$

(a) MinSum optimization

Minimize $\max_{I_j \in \mathcal{I}} C(S, I_j)$
1. $S_i \in \{0, \ldots, m\}$
2. $\sum S_i \leq m$

(b) MinMax optimization

2.3 Execution of algorithms with resource sharing

The last stage of the multi-selection is the concurrent execution of algorithms on the instance to solve, according to the computed resource sharing. All algorithms with a non null resource allocation are executed until one ends its execution. This requires to have interruptible algorithms. We will see however later that AMF provides a support to automate the interruption of algorithms in a concurrent execution on multi-core machines.

[3] This is deduced from the fact that we stop the execution on each instance as soon as one algorithm completes its execution

At this stage, we presented the multi-selection technique. The basic requirements of this technique (benchmark of instances, parallel or sequential algorithms, interruptibility, multi-core machines) makes it applicable on a large class of computational problem and machine architecture. We will discuss about the multi-selection technique in Section 5. In the next section, we illustrate how it is applied with AMF.

2.4 The multi-selection in AMF

The AMF framework is implemented in C/C++. The description of a computational problem in AMF is made through applications. An application has mainly a set of algorithms solving a same problem. It can be associated to a benchmark of instances for the related computational problem. AMF works like a problem solver that generates automatic combination of algorithms when it receives specifications of a benchmark and application. It also offers an interface for solving instances of known computational problems under the multi-selection.

In Figure 1a), we describe an example of application insertion in AMF. Applications and benchmarks are defined through through block of data of type AMF_Application and AMF_Benchmark. The application in the example is related to the CSP (Constraint Satisfaction Problem) and has three algorithms (max_deg, min_dom, max_dom_deg) that are CSP heuristics. These algorithms are sequential since the application is of type AMF_SEQ). Parallel algorithms can be also defined in AMF. In this case, the application is of type AMF_PAR. Certain requirements must be fullfilled for a successful application insertion. As indicated by Figure 1b), the algorithms solving the computational problem of the specified applications must be defined in the file AMF_AMF_ALG.cpp.

Figures 1c) and 1d) describe the insertion of a new benchmark for CSP. Similarly to the creation of an application, the insertion is made by invoking a function of the class AMF_Learner. Finally, in Figure 1f), we show how to update information for an inserted application.

Figure 1e) presents the resolution of an instance with AMF. All problem instances in AMF are specified using a block of data of type AMF_Instances. In the example, we defined an instance of the application CSP and used the function generateOneCSP for initializing data of the CSP problem. Then, we invoked an AMF solver (object of the class AMF_Solver) for its resolution under the multi-selection. After the resolution of the instance, the result is given as a void pointer that from a transtypage operation one can re-structure. All results are provided in AMF as an array of floats. The implementation of algorithms in AMF_AMF_ALG.cpp must take into account this information.

Figure 1 gives a tour of possible operations in AMF and manipulated objects. The internal structure of the framework is described in the next section.

3 The AMF component structure

Figure 2 describes AMF internal components and dependencies among them. The key component of this architecture is AMF_Learner that centralizes infor-

a) CREATION OF AN APPLICATION

```
#include "AMF.h"
int main(void){
 AMF_Application A;
 char[3][*] Alg = {"max_deg", "min_dom",
 "max_dom_deg"};
 A.app_name = new char[3];
 strcpy(A.app_name, "CSP");
 A.app_type = AMF_SEQ;
 A.alg_number = 3;
 A.Alg_name = Alg;
 AMF_Learner *L;
 L = new AMF_Learner();
 L->add_app(A);
 return 0;
}
```

b) REQUIRED CONTENT OF _AMF_ALG.cpp

```
//....others headers
void* max_deg(void *){//code };
void* min_dom(void *){//code };
void* max_dom_deg(void *){//code };
//....others headers
```

c) CREATION OF A BENCHMARK

```
#include<iostream>
#include "AMF.h"
int main(void){
 AMF_Benchmark B;
 AMF_Learner *L;
 L = new AMF_Learner;
 L->load_app_conf();
 B.app_id = L->getID("CSP");
 if(B.app_id >= 0){
  B.BenchFilename= new char[30];
  strcpy(B.BenchFilename,"CSPBenchmark");
  B.BenchReaderName = new char[30];
  strcpy(B.BenchReaderName, "CSPReader");
  B.benchsize = 150;
  L->load_bench_conf();
  L->add_bench(B);
 } return 0;
}
```

d) REQUIRED CONTENT OF reader.cpp

```
//....others headers
void* CSPReader(FILE *F){//code};
/* This function given a file descriptor
F towards data of CSP returns the next CSP
instance on which the pointer of F is*/
//....others headers
```

e) RESOLUTION OF AN INSTANCE

```
#include "AMF.h"
#include "AContainer/CSP/generateOneCSP.h"
using namespace std;
int main(void){
 AMF_Instances I;
 AMF_Solver *SOL;
 CSPInstance CSPInst;
 generateOneCSP(22, 6, -1, &CSPInst);
 float result[MAX_RESULT];
 I.app_name = new char[3];
 strcpy(I.app_name "CSP");
 I.arg = (void *)&CSPInst;
 I.mode = AMF_OFFLINE;
 I.objective = AMF_MINSUM;
 SOL = new AMF_Solver();
 SOL->solve(I, result);
 cout<< "the result is "<<
 (int)result[0] << endl;
 return 0;
}
```

f) UPDATE OF AN APPLICATION

```
#include "AMF.h"
int main(void){
 AMF_Application A;
 char[4][*] Alg = {"max_deg", "min_dom",
 "max_dom_deg", "max_deg" };
 A.app_name = new char[4];
 strcpy(A.app_name, "CSP");
 A.alg_number = 4;
 A.Alg_name = Alg;
 AMF_Learner *L;
 L = new AMF_Learner;
 L->load_app_conf();
 A.app_id = L->getID(A.app_name);
 A.app_type = L->get_app_type(A.app_id);
 L->update_app(A);
 return 0;
};
```

Fig. 1.: Possible usages of AMF

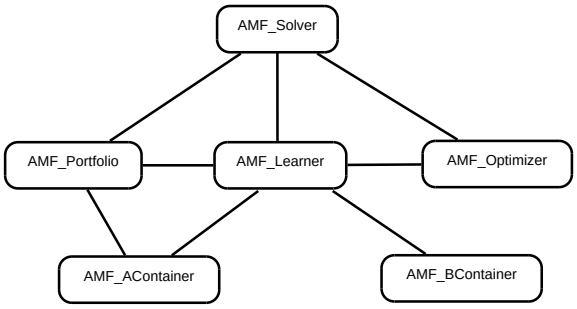


Fig. 2.: AMF Components and relations between them

mation about manipulated applications, available benchmarks and platform settings. The component AMF_Learner works as a tuning engine that learns from the physical architecture and generates automatic combination of algorithms for registered applications. It also works as a knowledge base informing other components about applications and benchmarks defined in AMF. AMF_Portfolio is responsible for the portfolio execution of defined combinations of algorithms. It communicates with AMF_AContainer that contains all algorithms solving computational problems of applications defined in AMF. AMF_BContainer contains multiple source files of benchmark instances used for tuning applications. Ideally, these instances must capture the difficulty of the related computational problem. The benchmark and algorithms containers can be modified by the user.

The submission of a request for the resolution of an instance is done in AMF through AMF_Solver. This component calls AMF_Optimizer for computing an adequate resource sharing and then run the a portfolio engine with the appropriate resource sharing. In the next sections we give details about these components.

3.1 The Solver

The solver component is constituted mainly by the class AMF_Solver. In this part we will present the following methods of this class:

```
void solve(AMF_Instance I, float *argout)
void set_MultiSelector(char *app_name, char *method_name)
```

The solve method takes in inputs an instance (I) and outputs an array of floats(argout) containing the solution of the instance. AMF_Instance is a structure whose fields comprise:

- the application name to which the instance refers,
- a void pointer toward the input data describing the instance,
- the mode of resolution chosen (online or offline),
- the time limit for instance resolution (this time is significant only if the chosen mode is online),

– the type of optimization (MinSum or MinMax).
– a proportion field $p \in [0, 1]$,

When the solve method is called, it communicates with the learner to have information about the referred application (in particular, the benchmark file tuned for its). Using this information, it will ask a resource sharing to the optimizer and will finally call a portfolio engine for its execution. Thus, the solver coordinates the entire execution of the multi-selection in AMF. It is important to notice that the solution of an instance in AMF is always returned as an array of floats. Despite this restriction, we believe that this format can handle many other internal representations (strings, integers etc.).

For the first phase of the multi-selection, AMF gives the possibility to define an appropriate method (a selector) for selecting a subset of candidate algorithms in the online mode of resolution. Each selector code must be defined in the file MultiSelector.cpp. The signature of a selector has the structure void <selector_name>(AMF_Instance I, int tab[], int k). Its implementation must ensure that <selector_name> modifies the array tab for indicating among the k algorithms available for the application, the ones selected (when tab[i] = 1, $1 \leq i \leq k$, the solver will consider that the algorithm i is selected).

After, the implementation of the selector, the AMF internal database can be informed of the new defined selector through a call to the method set_MultiSelector of the class Solver.

We believe that the definition of an efficient selector is a challenging task, requiring a good knowledge on the addressed computational problem. Some patterns for building selectors have been studied in [15, 2, 7]. Morever, AMF has a default one for all applications. Thus, one might have at a moment for a given application two selectors that can be used in the online mode. The choice between these selectors depends on the proportion field p of the AMF instance to solve. When an instance is submitted for the online resolution mode, if $p = 0$, then, the solver supposes that a personalized selector has been defined for the application and call it. If $p \in]0, 1]$, the AMF default selector will be called. This latter one will select randomly a proportion of p algorithms among available ones for the referred application.

In AMF, a constant array of functions containing addresses of selector is defined in the file MultiSelector.h. This pointer is used by the solver to determine for each application the associated selector. When the method set_MultiSelector is called, the content of MultiSelector.h is re-generated in order to update the pointer of selectors.

3.2 The Optimizer

The optimizer component is implemented by AMF_Optimizer. It is used in two main scenarios:

– When a new benchmark data is provided for an application, the optimizer is called by the learner to pre-compute an offline resource sharing;

– For the resolution of an instance in the online mode, the optimizer is called by the solver component to compute a good resource sharing within time limit.

The main functions used in the Optimizer are :

```
void getOnlineRS(AMF_Algp_desc)
void getOfflineRS(AMF_Algp_desc)
```

getOnlineRS serves to compute a resource sharing in the online mode and getOfflineRS serves for the offline mode.

The Optimizer structure The Optimizer components contains 4 classes: AMF_Optimizer, AMF_AO, AMF_MS, AMF_MMO. Dependencies between these classes are presented in Figure 3. The two classes AMF_MSO, AMF_MMO are

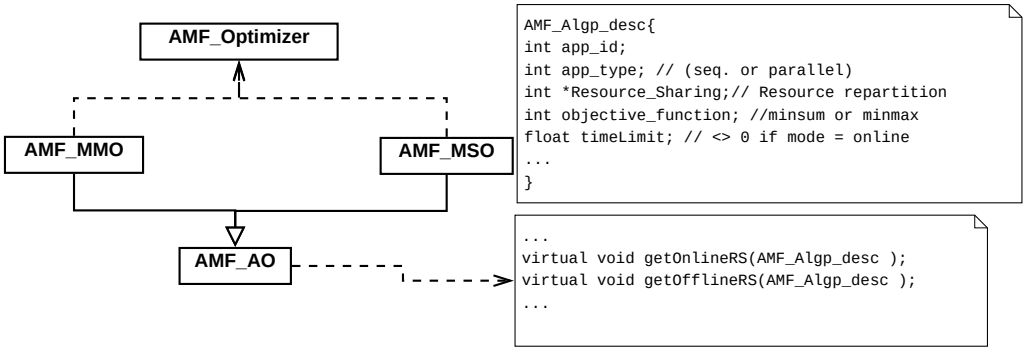


Fig. 3.: Classes of the optimizer component. The function getOnlineRS and getOfflineRS are designed to compute a resource sharing in the online and offline modes.

specialized on the computation of resource sharing under respectively the MinSum and MinMax objectives (see Section 2.2). These two classes are derived from an abstract optimizer class (AMF_AO). Finally, the class AMF_Optimizer works as interface of communication for other components.

Optimizer implementation *a) Main heuristics :* For computing resource sharing, AMF has for each optimization objective, 5 heuristics specialized in resource sharing building on parallel application and 5 ones for sequential applications. These heuristics solved the dRSSP model presented in Section 2.2 and are listed in Table 1. Detailed explanations on the heuristics implementations for parallel applications can be found in [5] and especially in [11] for HIF. Heuristics

Optimization heuristics in the parallel case ($\|\mathcal{A}\| = k$, $\|\mathcal{I}\| = n$ and m resources)		
Heuristics	Approx. ratio	Time complexity
HIF	arbitrary	$O(\min(k,m).(n^2k^2 + km^2))$
MAG	$k - g + 1$	$O(n2^{k-g}.(m+1)^g.(nk))$
MA	$2k - 1$	$O(k)$
RAND	arbitrary	$O(k)$
WTA	arbitrary	$O(nk)$
Optimization heuristics in the sequential case		
HIF^s	arbitrary	$O(m.n^2k^2)$
WTA^s	arbitrary	$O(nk)$
RAND^s	arbitrary	$O(k)$
OPT^s	1	$O\binom{k}{m}$

Table 1.: Heuristics used for optimization, guaranteed approximations ratio and complexity. k is the number of algorithms to combine, n is the size of the benchmark for tuning algorithms and m is the number of computational units in the dRSSP model.

for sequential algorithms are just adaptations of those of the parallel case where we limited the number of possible resources for each algorithm to 1.

We added small changes in the MAG implementation. The original one consists of selecting a number g of algorithms (guessed algorithms) on which all possible assignments of resources are explored. If for the g chosen heuristics, we explore an assignment of resources that use a total of m_g resources, one fairly shares the $m - m_g$ resources to the remaining algorithms (each algorithm then has approximately $\lfloor \frac{(m-m_g)}{k-g} \rfloor$ resources). One can easily notice that when the number of guessed algorithms $g = k$, the MAG heuristic gives the optimal solution.

We modified MAG in observing that given a selection of g algorithms, between the $k - g$ remaining ones, some might not have any resources in the exact solution. So, instead of assigning to all these algorithms $\lfloor \frac{(m-m_g)}{k-g} \rfloor$ resources, we considered any possible subsets of algorithms $k' < k - g$ algorithms among the remaining ones and shared between them the $m - m_g$ resources. The best resource sharing that we obtain is retained.

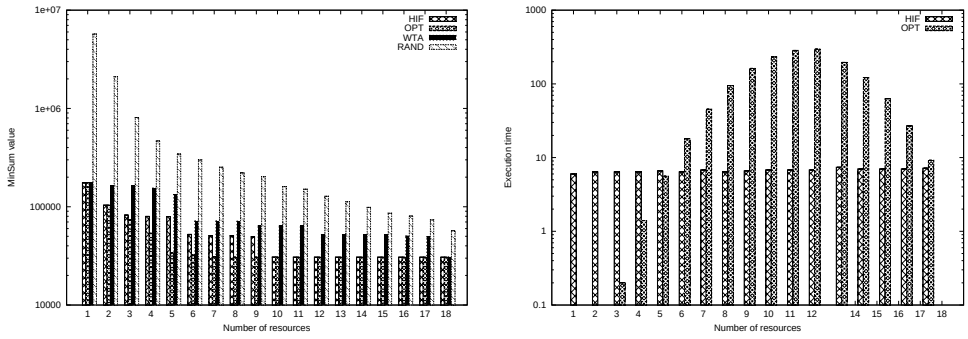
In the offline optimization mode (execution of getOfflineRS), the resource sharing is computed using the optimal heuristics (OPT^s and MAG with $g = k$). In the online mode (execution of getOnlineRS), the computation of the resource sharing must be done under the time limit defined in the AMF instance (see Section 3.1). Therefore, a tradeoff must be found between the quality of the solutions proposed and the time limit for the optimization.

b) Optimization in the online mode :

The online optimization mode comprises two steps: the construction of a *plan* that is, an ordered subset of optimization heuristics to be executed and the

execution of the optimization heuristics following the plan. The total time for deriving a resource sharing with a time limit t can be formulated as: $t_{rs}(I|t) = t_{cp}(I|t) + t_{ep}(I|t)$. In this expression, $t_{cp}(I|t)$ is the time required for computing the plan and $t_{ep}(I|t)$ is the time required for executing the plan. The construction of the plan must guarantee that $t_{ep}(I|t) \leq t$ and $t_{cp}(I|t)$ are small.

For having small values of $t_{cp}(I|t)$, we classified the optimization heuristics in three classes: the *polynomially fast* heuristics (MA, RAND, WTA), the *polynomially slow* heuristics (HIF), and the *exponential* heuristics (MAG, OPT). Given a time limit t, the construction of the plan starts with an estimation of the time required for executing and selecting the best resource sharing from the first class of heuristics. If the estimation suggests that this part of the plan will not exceed t, one evaluates the possibility of including *polynomially slow* heuristics with the remaining time limit estimated. Finally, *exponential* heuristics are considered. For this latter case, we search for the best value of g that will lead to an optimization under the remaining time limit.



(a) MinSum value of resource sharing computed by the heuristics

(b) Execution time required for computing the resource sharing

Fig. 4.: MinSum cost and execution time of heuristics for sequential optimization on a benchmark of SAT solvers

The clustering of optimization heuristics is motivated by their theoretical complexities (see Table 1) and experimental behaviors. In particular, Figure 4 depicts an experimentation made on a benchmark of sequential SAT solvers with the sequential heuristics for optimization. The results concern the MinSum objective. The experiments are done on a total of 23 sequential SAT solvers. We built on this set a resource sharing with optimization heuristics, assuming that we have $1, 2, 3..$ homogeneous resources. Details about the experiments can be found in [11]. The times of the *polynomially fast* heuristic are not reported but are always under 0.1 seconds. This figure exhibits the tradeoffs between the quality of the heuristic and the execution time required. For example HIF computes a better solution than WTA as shown by Figure 4a). However, it is

more time consuming as shown by Figure 4b).

3.3 The Learner

The learner is the central component of the AMF architecture. Its is mainly involved in the following scenarios:

- It learns platform settings (mainly at the installation of AMF) and tunes the analytical performance model of the optimizer;
- It is the main component for application and benchmark registration;
- The learner is also invoked by other components when they need information about applications (e.g. the optimizer needs to know if an application is parallel or sequential).

The basic functions used for these operations are:

```
void tune_bench(int app_id)
void add_app(AMF_Application)
void add_bench(AMF_Benchmark)
AMF_Application getData_app(int app_id)
AMF_Benchmark getData_bench(int app_id)
```

We will discuss them in what follows.

Tuning of the optimizer The tuning of the optimizer starts when the function add_bench is called. It consists of measuring the performances of heuristics listed in Table 1 on the parallel machine (we suppose multi-core) where AMF runs. The learner supposes that the execution time of each of each heuristics can be described as a real function $f(m, n, k, g)$ (m is the number of cores of the architecture). This choice is motivated by the complexity result obtained in Table 1. For the tuning, the learner explores the database of applications and benchmarks for finding possible value (n, k). For all valid points (n, k) [4], the learner considers all values of $g \in \{1, \ldots, k\}$ and makes multiple executions (actually 20 but it is customizable) of available optimization heuristics (HIF, MA etc). It retains the mean execution time obtained from the executions and save it.

Applications and benchmarks registration Benchmarks and applications registration are made in AMF through the learner functions: add_app and add_bench. An application to add is described through a block of data of type AMF_Application. This block is mainly characterized by:

- An application ID that is an integer unique to each application;
- A name which is supposed to be the computational problem to which we refer (e.g. SAT for Satisfiability);

[4] A point (n, k) is valid if there is a benchmark of size n for an application of size k

– A list of algorithms that are given through pointer toward algorithms implemented and available for the resolution of computational problem;
– A type that can be **Sequential** if all algorithms available for the application are sequential or **Parallel** otherwise. This information is important for the computation of resource sharing.

An AMF_Benchmark comprises mainly:

– An application ID that is the application referred by the benchmark;
– A benchmark source file that are brute instances representative of the computational problem;
– A benchmark reader that is a pointer towards a function that can extract an instance from the benchmark source file.

The registration of a new application will automatically create a unique identifier for it. It also inform the portfolio engine by code generation of this new registration (we will se how later).

To any application, there is associated a unique AMF_Benchmark block of information. For completing a benchmark registration, the learner first calls the function tune_bench for automatically generating a benchmark performance profile [5]. It then calls optimization heuristics that given this benchmark profile can compute an optimal resource sharing for algorithms known on the registered applications. Finally, it informs the benchmark container (by code generation) of the presence of a new identified reader function, and eventually tunes the optimization heuristic if a new couple *(number of benchmark instances, number of algorithms)* is introduced.

Communications with other components The learner is involved in multiple operations by other components when they need information about applications and benchmarks. To do so, they invoke its functions getData_app and getData_bench. To ease the access to this information, the learner maintains a table of applications and a table of benchmarks. These tables can be loaded explicitly as in Figure 1, in using the methods load_app_conf() and load_bench_conf().

The Learner manipulates a great deal of information related to the AMF internal setting. In Figure 5, we describe relations between information related to applications and benchmarks that are manipulated by the learner.

3.4 The portfolio engine

The portfolio engine is mainly invoked in two scenarios:

– It is invoked in the last stage of the multi-selection for the portfolio execution of algorithms;
– It is invoked by the learner when this latter has to generate a benchmark performance profile.

[5] This profile is given by the values $C(A_i, I_j, p)$ described in Section 2.2

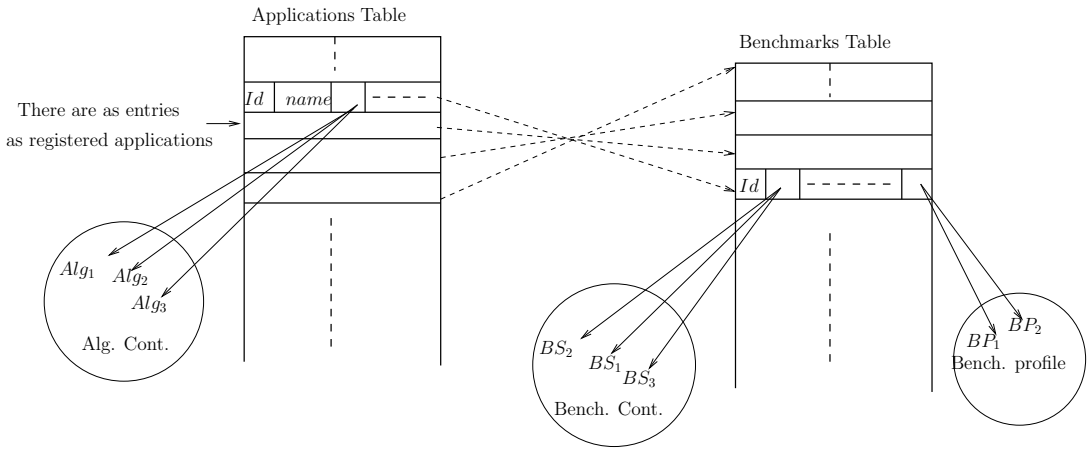


Fig. 5.: Data dependencies in the learner. Each entry of the application table is related to at most one entry in the benchmark table (when a benchmark is specified for the application). These entries also point to benchmark source files (in the benchmark container), benchmark performance profile files, and algorithms (in the algorithm container).

The portfolio engine keeps a pointer towards candidate algorithms (defined in the algorithm container) for application registered in AMF. The description of this pointer is in the file AlgPointers.h of the Portfolio container. This file is updated by the learner when an application is inserted or updated.

Given a defined resource sharing for an application, the portfolio engine can start the concurrent execution of algorithms accessible from its pointers following the resource sharing. In the concurrent execution, if the application is sequential, the resource sharing indicates the algorithms that will be run. In the parallel cases, it gives the number of resources for the execution of each algorithm. For coordinating the execution of multiple algorithms, the portfolio engine uses a Monitor object. This object has two important attributes:

- A boolean value that indicates that a solution is found;
- An array of float where can be written as result at the end of the execution.

These attributes can only be manipulated through synchronized functions implemented in the monitor. The monitor object is accessible in the file _AMF_ALG.cpp (where application algorithms are defined) and must be used in the implementation of algorithms to indicate that a result is found. We will see an example of utilization in Section 4.

3.5 Algorithms and Benchmark container

The algorithms and benchmark containers comprise implementation of algorithms and benchmark readers. Algorithms source codes in C/C++ must be implemented in the file _AMF_ALG.cpp. The signature of each algorithm must have the generic form void* <algorithmName>(void *). Despite the fact that the input argument is of type void *, its internal organization is of type AMF_Argument. This structure comprises:

- A pointer towards the input arguments that are data of the computational problem instance ;
- A pointer towards a monitor object used for the synchronization and collection of results;
- The number of resources for the execution.

For an effective coordination of the execution, each algorithm defined in _AMF_ALG.cpp must set in the monitor (given in input) a solution when it is found (An example of such an implementation is provided later). Moreover, AMF typically supports for now parallel algorithms based on threads.

The benchmark container comprises programs for reading benchmark source files provided in AMF. The definition of benchmark readers in AMF is done in the file reader.cpp of the Benchmark container. The signature of a reader must have the form void* <readerName>(FILE *F). The implementation must ensure that in the file pointed by F, a call of the reader function returns an instance as a void pointer. In the registration of a new benchmark, the name of the reader must be given.

4 Example of Constraint Satisfaction

We validated the AMF architecture for the resolution of the Constraint Satisfaction Problem. For this purpose, we inserted in AMF parallel and sequential versions of algorithms solving this problem. In Table 2, we detail the different actions that we did related to the parallel version of CSP (PCSP).

These actions define the general operations that are required for programming with AMF. As one can see the effort done by the developer consists mainly in providing algorithms, benchmarks, and benchmark readers in the algorithms and benchmarks containers. At the end of the execution of actions described in Table2, a new application is ready for utilization in AMF.

Parallel algorithms defined in AMF must be based on threads. This is important for the synchronization of the execution. Others rules in code writing must be respected in the actions done above, we will discuss about them in the Section 4.1.

4.1 Writing algorithms and readers

There are three aspects that are important while defining a new algorithm: the algorithm signature, the management of result and the coordination with

Action	Modified Components	Description
Definition of parallel PCSP algorithms	The files _AMF_ALG.cpp and _AMF_ALG.h of AMF_AContainer	The data structures for the PCSP instances are defined in _AMF_ALG.h and the algorithms in _AMF_ALG.cpp
Copy of a PCSP benchmark file in AMF	AMF_BContainer	A benchmark data file for PCSP is put in the Bcontainer
Definition of a benchmark reader	AMF_BContainer	the code of a reader for PCSP instance is written in the file reader.cpp
Insertion of the PCSP application	AMF_Learner, AMF_PContainer	In using the method add_app of the AMF_Learner, one inserts the PCSP application. It modifies the portfolio engine and the configuration of the application table
Insertion of a PCSP benchmark	AMF_Learner, AMF_BContainer	In using the method add_bench of the AMF_Learner, one inserts a PCSP benchmark. It generates pointers towards readers for the BContainer and configure the benchmark table
Global compilation	All components	The framework must be re-compiled to handle new codes generated in the previous steps

Table 2.: Defining a new application and benchmark in AMF

other algorithms. Let us suppose that data of PCSP instances as defined in the file _AMF_ALG.h are type CSPInstance. An example of algorithm for PCSP is described in Figure 6:

For receiving the inputs arguments, one must notice that AMF will execute the defined algorithms with an argument that is of type AMF_Argument (even if it is passed as a void pointer). This is what explains the cast operations that the developer must do at the beginning of Figure 6a). From the cast operation, the algorithm has access to a monitor (here std1→sync). It must use this latter one to set the result of its execution. The monitor provides the barrier functions (lock_barrier() and unlock_barrier()) to allow modifications of the result in mutual exclusion. Finally, throughout the function setEnd, ones indicates for all algorithms that a solution has been found. It is important when defining each algorithm for PCSP to think about setting the result in the monitor object when a solution is found. This is what will inform the other algorithm that are run concurrently to stop their execution.

When writing a reader, it is important to return a void pointer that can be translated from a cast operation in the adequate data structure. We give in Figure 6b) an example of reader structure for a CSP instance.

a) DEFINITION OF AN ALGORITHM

```
void* pmin_domain(void *arg){
  AMF_Argument *std1;
  std1 = (AMF_Argument *)arg;
  CSPInstance *std;
  int sat;
  std = (CSPInstance *)(std1->argin);
  // We solve std
  // A solution is found here and saved
  // in the variable sat
  std1->sync->lock_barrier();
  if(!std1->sync->is_ended()){
    float result[200];
    result[0] = (float) sat;
    std1->sync->setArg(result);
    std1->sync->setEnd();
  }
  std1->sync->unlock_barrier();
}
```

b) DEFINITION OF A READER

```
void* PCSPReader(FILE *F){
 CSPInstance *Inst;
 Inst = new CSPInstance;
 // read the last un-read instance of F
 return (void *)Inst;
}
```

Fig. 6.: Example of algorithms and benchmark reader

4.2 Qualitative assessment

Using AMF, we defined one sequential and parallel CSP applications, each having with 9 algorithms. The description of these algorithms can be found in [11]. For each application, we associated a benchmark of 225 CSP instances described in [11]. Then, we consider a scenario in which one has to solve again these 225 instances using AMF. In this scenario, we used a selector that ignores the benchmark instances. Thus, even if AMF has been tuned on the instances that we have to solve, it does not use this information.

In this scenario, we evaluated both the offline and online mode of resolution and the resolution under different optimization objective. We run the experiments on a parallel multi-core machine with 4 cores. The cores have a frequency of 2661MHz and hyper-threading is used in each core.

Experiments in the offline mode Table 3 presents the execution time obtained with AMF for solving the CSP instances with sequential algorithms. In this table, p is the number of threads that are created in the resolution. One can see here that if 4 threads are used in the MinSum optimization, the execution time obtained is better than those of the best single algorithm. Execution times presented are the means that we obtained from 30 executions. The standard deviation in the time was small (lower than 0.002).

We present in this table both theoretical estimations made by the AMF optimizer when computing a resource sharing and experimental results observed for the resolution of instances under the MinSum and MinMax objectives. Theoretical predictions do not coincide with measured execution times. This is due to cache sharing and hyper-threading overhead in the concurrent execution. The hyper-threading overhead is clearly visible since the more the number of threads

exceeding the number of cores, the more the difference with theoretical estimations. Finally, the time for the MinMax optimization here does not change

($\|\mathcal{A}\| = 9$, $\|\mathcal{I}\| = 225$ and $m = 8$)				
	Theoretical		Experimental	
p	Th. MinSum	Th. MaxSum	MinSum	MinMax
1	526	31	526	31
2	435	31	443	31
3	419	31	431	31
4	407	31	423	31
5	402	31	438	31
6	402	31	538	31
7	402	31	655	31
8	402	31	723	31
Best single algorithm, MinSum = 526, MaxSum = 31				

Table 3.: Execution time (in seconds) in the offline resolution mode for sequential CSP algorithms. p is the number of threads.

because the CSP algorithms are run with a maximal time unit of (30 seconds) and, there are some instances that can not be solved under this time limit.

We also did an evaluation of the offline optimization with parallel CSP algorithms. The results are reported in Table 4.

($\|\mathcal{A}\| = 9$, $\|\mathcal{I}\| = 225$ and $m = 8$)						
	Best alg.		Theoretical		Experimental	
p	Best MinSum	Best MinMax	Th. MinSum	Th. MaxSum	MinSum	MinMax
1	526	31	526	31	526	31
2	190	31	190	31	190	31
3	70	30	70	30	70	31
4	60	6	38	6	41	7
5	53	6	27	6	30	7
6	60	6	21	4	27	5
7	65	6	19	4	32	5
8	53	6	19	4	32	5

Table 4.: Execution time (in seconds) in the offline resolution mode for parallel CSP algorithms

This evaluation again shows that AMF can outperform the best algorithm.

However, when we have more threads than core, hyper-threading overhead is more present.

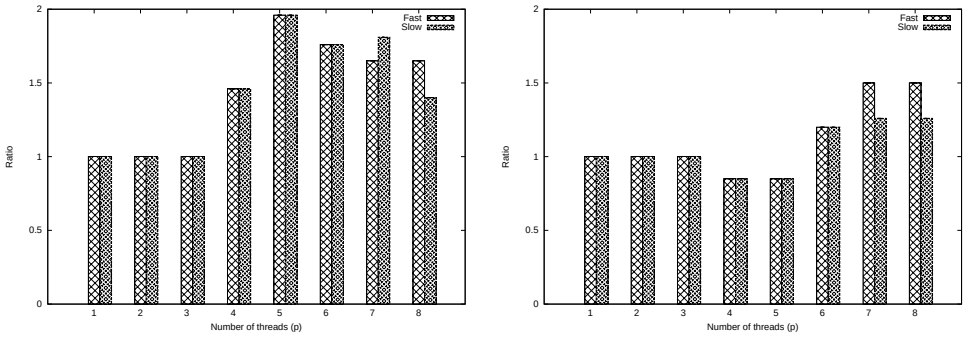
Finally, in Table 5 we depict the cumulative execution time that was needed for generating a benchmark profile for CSP instances and for computing an optimal resource sharing (given a benchmark profile) in the offline mode. The optimization time is not important mainly because we do not have many cores (see Table 1 on complexity analysis). The tuning time however is greater and, could have been more important if we did not decide to execute parallel and sequential CSP algorithms only one time instead of 20 times, when from the first execution, it is clear that the algorithm can not solve the instance under the 31 seconds.

Sequential		Parallel	
Optimization	Profiling	Optimization	Profiling
1 s	23 hours	4 s	160 hours

Table 5.: Times for Benchmark profile generation and computation of optimal resource sharing

Experiments in the online mode In these experiments, we solved CSP instances with different time limits in order to have a solution under a *polynomially slow* and *polynomially fast* optimization (see Section 3.2). In the first phase of the multi-selection, the chosen selector selects all algorithms available for the application. There is no interest in having such a selector for online optimization. But, we used it solely to have a setting of comparison with the offline optimization. In Table 7 we depict for solving the 225 instances, the ratio between the cumulative execution time with AMF under online optimization and the optimal measured times when performing offline optimizations. The evaluation is done with the parallel CSP application.

While $p \leq 3$, there is no overhead incurred by online optimization. This is due to the fact that the optimal solution consists in executing the best algorithm. This solution is found by the WTA heuristic for optimization. When p is between 4 and 6, the best algorithm is not the optimal possible solution. However, the polynomially fast and slow optimization choose a plan that cede all resources to the best algorithm. When $p = 7$, a difference appears between the polynomially fast and slow optimization modes. While the former sets all resources to the best algorithm, the latter suggests a better resource sharing. However, the polynomially slow resource sharing leads, in the execution, to more overhead than the polynomially fast ones. Finally, on $p = 8$, the polynomially slow optimization computes a resource sharing that in the execution outperforms the polynomially slow optimization.



(a) MinSum case (b) MinMax case

Fig. 7.: Ratio between the cumulative execution time obtained with online optimization and the one obtained with offline optimization.

These experiments show that the multi-selection technique as implemented in AMF can effectively combine multiple algorithms solving the same problem in order to reduce the execution time needed in problem resolution. They also show that there is a difference between the expected time in AMF when making combination of algorithms and the effective times observed in practice. Therefore, there is a need to integrate more knowledge about machine architectures (for example hyper-threading overhead) in AMF. We have shown experimentally that the multi-selection technique is efficient in combining multiple algorithms solving the same computational problem. In the next section, we provide a short analysis of the technique. The objective is to provide general intuitions that justify its utilization.

5 Analysis of multi-selection

We discuss here the advantage of multi-selection as done in AMF with respect to the importance of selecting more than one algorithm for solving a problem instance. We also present an alternative model: algorithm ranking [20, 3]. In all the discussion, we consider that we have a shared memory parallel context with a finite number of m homogeneous units of computations.

a)Multiple selection vs unique selection: Let us suppose that for solving an instance with some candidate algorithms $\mathcal{A} = \{A_1, \ldots, A_k\}$ we select a single algorithm. Let us also suppose that we have n instances to solve. On each instance i, let us denote by t_i^* its minimal resolution time with one algorithm of the set $\mathcal{A}$

Given a technique T_x, the mean expected time for solving the n instances is denoted $E[T_x(n)]$. The risk of this technique is denoted $E[T_x(n)] - t_{opt}$ where

$t_{opt} = \sum_{i=1}^{n} t_i^*$.

When selecting a unique algorithm, we might have a probability of p for selecting the right algorithm. For each instance i, let us denote by $t_i^1, \ldots, t_i^k$ the time required to solve it respectively by $A_1, A_2, \ldots, A_k$. Let us also assume that we have an equiprobability of having any algorithm as the wrong selected ones. The mean time for solving n instances by selection of a single algorithm is then $E[S(n)] = \sum_{i=1}^{n} p.t_i^* + (1-p)(t_i^* + \frac{1}{k}\sum_{u=1}^{k}(t_i^u - t_i^*))$.

For solving all instances with a multi-selection of k algorithms, we can expect a time of $E[M(n)] \leq \sum_{i=1}^{n} \alpha_i t_i^*$ (since we execute all algorithms concurrently) where α_i depends on the resource sharing [6]. When isolating the optimal resolution time $t_{opt} = \sum_{i=1}^{n} t_i^*$, we have $E[S(n)] = t_{opt} + \frac{(1-p)}{k}\sum_{i=1}^{n}[\sum_{u=1}^{k}(t_i^u - t_i^*)]$ and $E[M(n)] \leq t_{opt} + \sum_{i=1}^{n}(\alpha_i - 1)t_i^*$.

It is reasonable to bound the value of α_i with, for example, the number of resources if there is a linear parallelism, and we have less algorithms than resources $k \leq m$. We will then have $\alpha_i \leq m$. Thus the risk in offline multi-selection can be bounded at a fixed distance factor to the optimal solution while the quantity $\frac{(1-p)}{k}\sum_{i=1}^{n}[\sum_{u=1}^{k}(t_i^u - t_i^*)]$ can be arbitrarily large. This means that the selection of a unique algorithm is more risky than the multiple selection in the offline mode.

Smart values of α_i could be proposed to minimize $M(n)$. In particular, we can share resources to algorithms in order to tolerate, an important overhead on instances whose execution time is small for all algorithms. This is the key point of heuristics for optimizing resource sharing in AMF.

The bigger the value of p, the smaller the risk in unique selection. This is the main interest for an online multi-selection. Indeed, if it is possible to detect with high probability what is the optimal algorithm, then, it might be possible to have a process that can choose for each instance i a subset of k_i algorithms ($k_i \leq k$ and $\sum_{i=1}^{n} k_i < nk$) such that the best algorithms on the instance is included on the subset with a probability of 1. Thus, the expected time for the portfolio will be $E[M(n)] \leq \sum_{i=1}^{n} \beta_i t_i^*$ and since $k_i \leq k$, we have less algorithms executed concurrently and we could expect that $\sum_{i=1}^{n}(\beta_i - 1)t_i^* \leq \sum_{i=1}^{n}(\alpha_i - 1)t_i^*$.

b)Algorithm portfolio vs algorithm ranking: In algorithm ranking, the selected algorithms are not executed concurrently. A fixed amount of time or cutoff and a ranking between algorithms is decided. Then each algorithm is executed on the instance to solve during the cutoff time decided and following the decided ranking. The executions is stopped when a solution is found.

This model of execution is certainly a good alternative to algorithm portfolio. One advantage is that there is no need to compute a resource sharing since each algorithms is executed with all resources. Algorithm ranking has been used with interesting results in [20].

In algorithm ranking it is important to fix a cutoff time. This is not easy since if the cutoff is two small, then on some instances, it might be impossible to have a solution under it. Let us suppose that the cutoff has a value of t and that there

[6] We suppose that all algorithms stop immediately when a solution is found

is at most one algorithm that can solve each instance under this cutoff. Given a ranking of algorithms, an instance will be solved by the first algorithms or if not, the second and if not the third etc. We suppose that any instance has an equiprobability p to be solved at each rank. Thus, the time for solving n instances will be $E[R(n)] \geq \sum_{i=1}^{n}(pt_i^* + p(t + t_i^*) + p(2t + t_i^*) + \cdots + p((k-1)t + t_i^*))$. In isolating the optimal resolution time, we have $E[R(n)] \geq t_{opt} + p\sum_{i=1}^{n} \frac{k(k-1)}{2}(t_i^* + (t - t_i^*))$. The risk again depends on the cutoff factor. In order to guarantee that each instance will be solved under the cutoff, this value must be in general big, we find the algorithm portfolio approach less risky.

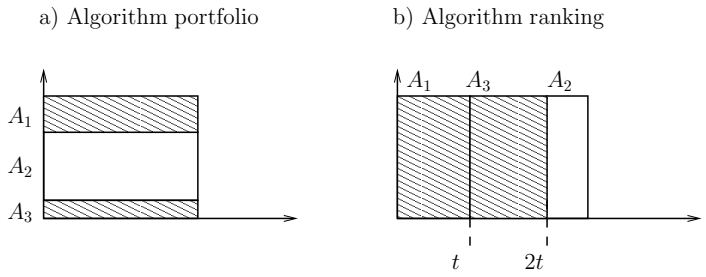


Fig. 8.: Example of execution pattern in algorithm portfolio and algorithm ranking. In the case of portfolio, all algorithms are executed concurrently to solve the instance while they are executed given a rank and under a time limit (here t). In both cases, we have useless executions (dashed in the figure).

The multi-selection technique is a less risky approach when there is an important difference between execution of algorithm. This is in particular of heuristics solving hard computational problem.

6 Related Work

The main philosophy that has been considered in the automation of algorithms combination is the problem specific approach. In such approaches, an adaptive algorithm that can adapt its execution depending on the machine architecture is designed for a specific computational problem. Given a computational problem $\mathcal{P}$, the adaptive algorithms manage a pool of candidate algorithms designed for $\mathcal{P}$. Depending on the machine architectures and instance of $\mathcal{P}$ to solve, it selects the most appropriate algorithm(s) to obtain good performances. To be able to make these choices, the adaptive algorithm will learn how to proceed during its installation on each platform from a finite benchmark of $\mathcal{P}$ instances. This problem specific approach has been applied successfully on many computational problem such as matrix multiplication [17, 9], Sorting [1, 19, 4], Fast Fourrier Transform [19, 14], etc..

Let now consider the problem specific approach philosophy in the perspective of evolution of algorithms and machine architectures. We can say that if only the machine architectures change, the philosophy of adaptive algorithms suggests that we might not necessarily need to re-design the problem specific approach (since the algorithm adapts itself to the platform). However, if the set of known algorithms for the problem changes, a design of adaptive algorithms is required in order to include this new algorithm in the suite (otherwise, it is possible that there exists a more efficient external algorithm than the adaptive algorithm). However, one cannot anticipate the algorithmics evolution on a computational problem. Moreover, depending on the utilization context, the set of algorithms required for solving a problem can change. For example, there is no advantage of using the quicksort algorithm in a context where there are only a small number of item to sort [13]. To deal with this, one can observe how parallelization is done in parallel computation. Parallel programming proposes both problem specific library (on sorting, searching etc.) and general API like MPI and pthreads for simplifying the implementation of parallel program. Considering this example, we can say that the design of automatic combination of algorithms also requires general API that eases the implementation task without being specific to a particular computational problem. This point of view has received an increasing interest over the last decade.

Among the most relevant studies, let us recall the AEOS method [8] used on Self Adaptive Numerical System [10]. AEOS deals with automatic selection between multiple implementations of the same algorithm (in changing for example the order of the loops in the implementation). AEOS has been used in particular as a methodology for tuning and selecting kernels on dense and sparse linear algebra problems.

In [18], a framework is proposed for composing a general parallel algorithm with sequential algorithms in order to automatically balance the load during the parallel execution. Such a solution is typically well suited when there are parallel algorithms based on divide and conquer with few communications. In [6], a framework (mainly conceptual) for dynamic adaptation of parallel codes (in a context of computational grid) is proposed.

The works that are certainly the closest to the contribution proposed of this paper are those done on hyper-heuristics [7]. The idea is to develop generic search procedures that work on a space of algorithms solving the same problem. Typically, this search must select the most efficient algorithm solving a computational problem. This idea has been validated on many case studies like the resolution of time tabling problem [16].

In this work, we have proposed a framework for developing adaptive and parallel programs based on automatic combination of algorithms. The proposed solution is based on the multi-selecttion and can be considered as a first step towards the development of parallel hyper-heuristics.

7 Conclusion

We presented in this paper a new Framework for the design of adaptive libraries of algorithms. AMF is based on a collaborative approach allowing the users to constantly improve their knowledge on the resolution of a target problem. The adaptation is done through the multi-selection technique. The key point of this technique is the computation of efficient resource sharing for running concurrently a set of algorithms on an instance to solve. We described the implementation of this technique in AMF and provided a qualitative assessment on the Constraint Satisfaction Problem.

For continuing this work, our first envisioned issue is to extend AMF for distributed contexts. A first step for reaching this objective consists of introducing in AMF a support for executing algorithms designed based on multiple operating system processes execution. We believe that the main challenge for this purpose is in the distributed coordination of concurrent execution of algorithms. For the moment, we operated the coordination at the algorithmic level throughout a monitoring. For supporting parallelism at a process level, this variable can be mapped in a shared memory space accessed by all processes. Another option is to operate the coordination of execution at the operating system level. In this case, we will check the state of execution of launched processes and then stop all processes when one solution is found. A system coordination might also have the advantage of avoiding the necessity of manipulating synchronization informations while writing algorithms in the file _AMF_ALG.cpp.

Another interesting issue is to propose efficient generic selectors for the first phase of the multi-selection. We showed that for solving an instance under multi-selection, there is a compromise to make between the number of algorithms selected and the overhead in the concurrent execution. In employing the recent techniques of multi-objective optimization, good algorithms selectors can be proposed for the first phase. Finally, we believe that the computation of the plan in the optimizer may also be improved. Indeed, for instance this problem is closed to the knapsack problem for which efficient heuristics already exists.

References

1. P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel C++ library. *Lecture notes in computer science*, pages 193–208, 2003.
2. S. Bhowmick, V.Eijkhout, Y.Freund, E. Fuentes, and D. Keyes. Application of machine learning to the selection of sparse linear solvers. `www.tacc.utexas.edu/~eijkhout/Articles/2006-bhowmick.pdf`.
3. Sanjukta Bhowmick, Lois C. McInnes, Boyana Norris, and Padma Raghavan. The role of multi-method linear solvers in pde-based simulations. In *ICCSA (1)*, pages 828–839, 2003.
4. Eran Bida and Sivan Toledo. An automatically-tuned sorting library. Technical report, School of Computer Science, Tel-Aviv university, 2006.

5. M. Bougeret, P.F. Dutot, A. Goldman, Y. Ngoko, and D. Trystram. Combining multiple heuristics on discrete resources. In *11th Workshop on Advances in Parallel and Distributed Computational Models APDCM, (IPDPS)*, 2009.
6. Jérémy Buisson, Françoise André, and Jean-Louis Pazat. A framework for dynamic adaptation of parallel components. In *PARCO*, pages 65–72, 2005.
7. Edmund K. Burke, Mathew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John R. Woodward. Exploring hyper-heuristic methodologies with genetic programming. In Christine L. Mumford and Lakhmi C. Jain, editors, *Computational Intelligence*, volume 1 of *Intelligent Systems Reference Library*, chapter 6, pages 177–201. Springer, 2009.
8. J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R.C. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293 –312, feb. 2005.
9. J. Dongarra, G. Bosilca, Z. Chen, V. Eijkhout, GE Fagg, E. Fuentes, J. Langou, P. Luszczek, J. Pjesivac-Grbovic, K. Seymour, et al. Self-adapting numerical software (SANS) effort. *IBM Journal of Research and Development*, 50(2-3):223–238, 2006.
10. Victor Eijkhout, Erika Fuentes, Thomas Eidson, and Jack Dongarra. The component structure of a self-adapting numerical software system. *Int. J. Parallel Program.*, 33:137–143, June 2005.
11. Alfredo Goldman, Yanik Ngoko, and Denis Trystram. Optimizing resource sharing on cooperative execution of algorithms. Technical report, University of Grenoble, 2011.
12. B.A. Huberman, R.M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51–54, 1997.
13. Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley second Edition, 1998.
14. Steven G. Johnson Matteo Frigo. FFTW: An adaptive software architecture for the FFT. In *proceedings of the International Conference on Acoustics, Speech and Signal Processing*, Seattle, Washington, May 1998. ACM SIGARC.
15. Y. Ngoko and D. Trystram. Combining numerical iterative solvers. In *PARCO*, pages 43–50, 2009.
16. Rong Qu, Edmund K. Burke, and Barry McCollum. Adaptive automated construction of hybrid heuristics for exam timetabling and graph colouring problems. *European Journal of Operational Research*, 198(2):392–404, 2009.
17. Antoine Petitet R. Clint Whaley and Jack Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.
18. Jean-Louis Roch, Daouda Traoré, and Julien Bernard. On-line adaptive parallel prefix computation. In *Euro-Par*, pages 841–850, 2006.
19. María Jesús Garzarán Xiaoming Li and David A. Padua. A dynamically tuned sorting library. In *proceedings of the 2004 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 111–124, San Jose, California, June 2004. IEEE Computer Society.
20. Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for sat. *J. Artif. Intell. Res. (JAIR)*, 32:565–606, 2008.

This article was processed using the LaTeX macro package with LLNCS style